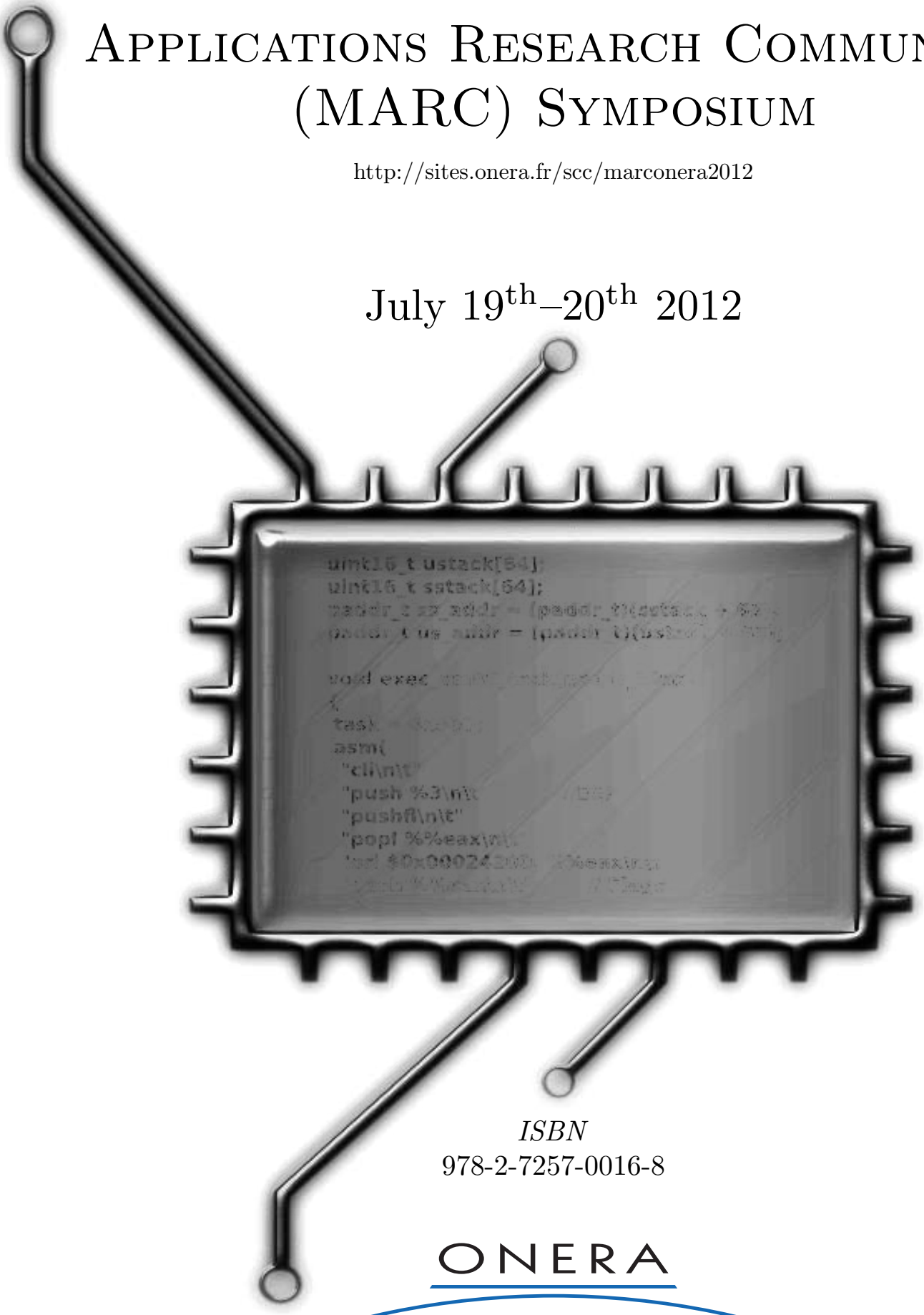
Interactive Visual Task Management on the 48-core Intel SCC

Jimi van Der Woning, Roy Bakker

Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium

http://sites.onera.fr/scc/marconera2012

July 19th–20th 2012



ISBN
978-2-7257-0016-8

ONERA
THE FRENCH AEROSPACE LAB

 ISBN: 978-2-7257-0016-8

Interactive Visualization and Task Management on the 48-core Intel SCC

Jimi van der Woning and Roy Bakker
Informatics Institute, University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam, The Netherlands

***Abstract*—In this paper we propose and describe how we have built a tool that enables a user to interactively monitor and manage a many-core system like the 48-core experimental Single-chip Cloud Computer (SCC), which was created by Intel Labs targeting the many-core research community. We provide the user with a visual representation of the current state of the system on multiple levels of detail, such as chip, core and task. We allow the user to create, start, pause and migrate tasks across different cores. We also allow the user to easily adjust the voltage and frequency of the chip. However this tool can run on any PC with a screen and input devices, we have optimized the interface to run on a multi-touch device for the best ease of use.**

I. INTRODUCTION

The Single-chip Cloud Computer (SCC) experimental processor [1] is a 48-core *concept vehicle* created by Intel Labs as a platform for many-core software research. It provides an on-chip message passing network, a non cache-coherent off-chip shared memory and dynamic frequency and voltage scaling.

Unlike currently available multi-core systems, the SCC is an on-chip distributed system. Even though efforts are already made and still continuing on writing and porting operating systems or virtualization layers that can manage the chip as a whole [2], the most common use of the SCC system is currently to have every core managed by its own instance of a slightly modified version of the Linux kernel. As a consequence, it gets harder for users to gain insight in the current state of the system. Also, it is not trivial to map tasks on the system while keeping the load balanced and energy consumption low, without a complete understanding of the state of the system.

We propose a management system for independent many-core systems like the SCC, which enables users to interact with the system. The user of the system must be able to:

1) Monitor the system:
   - Current load, state and power consumption of the chip.
   - The current resource usage for each core.
   - The resource usage per monitored process on a core.
   - Task output.
   - Overview of running, waiting, completed and possibly failed tasks.
2) Manage the system:
   - Easily create a task.
   - Start a task on a single specified core, or the best core available (suggested by the system).
   - Migrate tasks to other cores, either manually with or without suggestions from the system, or potentially automatically.
   - Control frequency and voltage.

All of the above can easily be controlled from a user interface running on a regular PC with mouse interaction, but to improve the user experience and ease of use even more, the system is optimized for use with a multi-touch system.

Manually managing a system like the SCC does not seem to be very efficient in daily practice. For this purpose, it is better to use an automated grid- and cluster management system. However, we think that our system is very useful in both research and education. It can mainly be used for experimenting with task placement and voltage and frequency settings, while having a clear understanding of what is currently happening from a user friendly interface. The system can later be attached to an automated task manager or grid engine.

In this paper we will discuss our experiences in building such a management system for the SCC. We assume sufficient knowledge of the SCC architecture and its memory system, as this is broadly covered by both related work [1], [3] as well as our previous work [4], [5]. Some related work on visualization, but not on cluster management systems, is described in Section II. We discuss which approaches and tools we have used for the implementation in Section III. In Section IV we evaluate these different approaches and we conclude with a discussion and future work in Section V.

II. RELATED WORK

Since the beginning of the SCC research, a *Performance Meter* has been available as part of the *sccKit* provided by Intel (See Figure 1). This tool shows us the current load for all the independent cores, as well as the overall usage and power consumption of the system. It is a nice tool, but it will not be sufficient for our proposed system, as there is no way to actually manage the SCC.

QNX Software Systems [6] has developed software for the visualization of many-core applications, which is used mostly for the development of applications in embedded systems. It is not considered for the management of a running system. Other visualization frameworks such as IBM's Tuning Fork [7] are available to monitor the performance of a system. Many visualization tools only operate on traces instead of a running system. We have not found a system that combines visualization and management on the same high level as we want to.

Figure 1: The SCC performance meter (`sccPerf`).

III. MANYMAN - THE MANY-CORE MANAGER

The many-core visualization and management tool has been dubbed *ManyMan*, as in Many-core Manager. ManyMan consists of two main parts, a front- and a back-end, with a communication layer in-between. The reason for this separation is twofold. First, the front-end could theoretically be attached to a different many-core chip, or, the other way around, a different front-end could be attached to this back-end. This increases the usability of the tool, since it is not restricted to just one chip or interface. The second reason for this separation is of a more practical kind. At the University of Amsterdam, the SCC is located in a server room where no monitor could be easily attached to it.

BACK-END

The back-end has been written completely in Python, for which there are two main reasons. First, Python is a relatively easy programming language which allows for rapid development. Second, Python provides good support for running many threads, starting shells on remote machines using subprocesses and TCP communication. The back-end performs multiple tasks:

A. Monitoring

Monitoring the chip is half of the visualization and management tool. One would like to know the status and payload of every core and task on the system. Unfortunately, the SCC does not provide such information about the chip as a whole, which means that it needs to be retrieved separately from each core.

In order to access one of the SCC's cores, one needs to open an SSH connection to the core in question from the MCPC. Since an SSH connection needs to be opened for each task that runs on the SCC, SSH Connection Sharing is used. This mechanism allows us to have only one TCP connection to an SCC core, with one-time authentication. This master connection has to be active while all subsequent connections are, which makes the monitoring process the perfect candidate to be that connection.

In order to obtain the payload information of each core, the Unix `top` command is used. At adjustable intervals, `top` provides information about all processes that are running on the core and the total payload of the core itself. The fact that this information is everything that has to be shown, makes `top` the ideal monitoring solution. However, as `top` accesses more information than just that information that is needed, it might create some overhead.

It has to be noted that any resource usage of processes that have not been started using ManyMan, as for example kernel processes, will be marked as overhead. This overhead will be visible in the total core payload, but, of course, not in the per task payload. Because of that, the task payloads will not add up to the total core payload.

B. Task creation

When a task needs to be created, a child connection is added to the monitoring master SSH connection of the core on which the user has decided the task should run. On this child connection, the task is started with BLCR's `cr_run` command, and a small wrapper that enables us to obtain the process identifier (PID) of the task on the remote core. When the program starts to run, its output will be buffered. It will be sent to the front-end upon request.

In case a user does not know which core to start a task on, a *smart-start* function has been implemented. When smart-starting a task, the core with the least CPU and memory usage is selected. In this process, both the CPU and the memory usage have the same weight. A possible growth in CPU or memory usage is foreseen by also taking the number of running tasks on a core into account. The more tasks are running on a core, the smaller the chance a task will start there gets. As soon as the best core to run the task on is found, the task will be started on that core as usual. Note that this smart-start function does not keep track of the history of core usage or whatsoever, but it just looks at the current core state.

C. Task migration

To enable the migration of tasks, we make use of the Berkely Lab Checkpoint/Restart library [8]. Using BLCR, tasks can easily be stopped (checkpointed) and restarted later while releasing all resources. Restarting a task can also be done on other cores or even other compatible machines. Besides the benefits, checkpointing also creates overhead, as the BLCR library writes the complete state of the process to the filesystem. A task that needs to be migrated will first be checkpointed using the `cr_checkpoint` command from the BLCR library. The location of the context file that is hereby created will be stored in order to be able to restart the task later. When checkpointing is complete, the task can be restarted on

the desired core using `cr_restart`. When a user does not want to restart the task yet, it will instead be moved to the list of waiting tasks. Since all cores of the SCC mount the same `/shared` directory, one does not have to worry about sending context files among cores. These can just be found on the exact same path as where they were originally stored. At restart, the `cr_restart` command will be executed with the `-no-restore-pid` flag to avoid PID collisions on the remote core.

Just like the smart-start function, a *smart-move* function has been implemented. Using this function, a task can be moved to the best possible core, which is found the same way as it is done when smart-starting, except for the fact that the new core will never be the core the task is already running on.

D. Task pausing / resuming

Since checkpointing a task produces some overhead, tasks can also be paused using the traditional POSIX STOP signal, after which they can be resumed by sending the POSIX CONT signal. It has to be noted that even though the process is paused and will not use the CPU, it will not release resources such as memory. Due to this fact, paused tasks cannot be moved to other cores, as long as they are not checkpointed. Besides that, not releasing memory might be a problem for the SCC cores, since their private memory is limited (around 640MB). For manually scheduling CPU intensive tasks however, this is a great solution.

E. Communication

The communication between front- and back-end makes use of a TCP connection. The connection is currently open and not encrypted as both client and server are within the same access restricted network. When the server needs to be available on the public internet, some form of authentication has to be implemented. Across the TCP connection, messages are sent in the JSON format. This format is human-readable, which allows for easy debugging and portability.

FRONT-END

The GUI part has been optimized for use with a (multi)touch display. For the front-end we make use of *Kivy* [9]. Kivy is an open source library for rapid development of applications that make use of innovative user interfaces, such as multi-touch applications. The same Kivy source code runs on Linux, Windows, MacOSX, Android and IOS, providing us with the best flexibility.

F. Chip overview

The chip overview as a whole is shown in figure 2. In the middle part, the cores are presented in the order they are physically arranged at on the chip. This order serves no functional purpose, but has been chosen to provide the user a realistic view of the chip and provide the same layout as the `sccGui` does. The payload of each core is visualized by a coloured overlay that changes in both colour and size. The portion of the core that is taken by the overlay literally

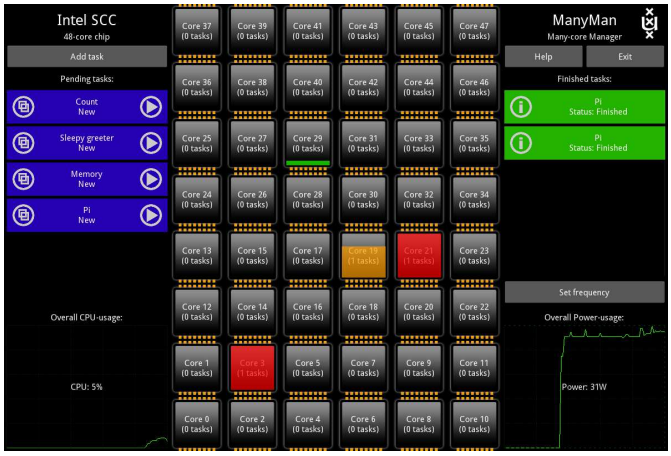


Figure 2: ManyMan's main window, the chip overview.

translates to the core's CPU usage, where the colour changes from green at 0% CPU to red at 100% CPU. Although the information is only updated once a second, the overlay is animated to fade to the new payload within that second. The main window also notes the number of tasks per core.

On the left side of the window, a list of tasks is shown. These tasks are currently not running on any core, but are either not started yet (*new*) or have been stopped by the user (*stopped*). In order to (re)start such a task, one can simply drag it to the core he or she wants it to run at. If the user does not care on which core the task will run, the task may be *smart-started* by tapping the play button on the right side of the task. By clicking the copy button on the left side of a task, the task can be duplicated. This is especially useful when said task is a benchmark program.

The right side of the window provides a list with tasks that are either finished or have failed to complete. In this list, the task's output and statistics can still be viewed, but it can not be dragged any more. The delete button will remove the task completely from the ManyMan system.

Figure 3: ManyMan's core view, containing information about core 28.

G. Core view

When tapping a core in the chip overview, a popup like the one in figure 3 will open. On the left side, the history of the CPU and memory usage is shown. In these graphs, the white line indicates the total load of the core, which consists of all tasks started by ManyMan, plus all overhead (i.e tasks not started through ManyMan, and OS overhead). The coloured lines in the performance graphs indicate the payload of the tasks that have been started using the many-core management system. The colours of these lines match the colours of the tasks in the task list on the right side of the core view.

The tasks in the task list can be moved to a different core by simply dragging them to the core a user wants them to run on. When dragging a task, all open core popups will swerve out of the way so that they do not block any core. The *smart-move* option, along with some additional controls and information, is located in the detailed task view. This task view can be opened by tapping the information button on the left side of a task. When a task is dragged to anything that is not a core, as for example the task list on the side of the main view, it will be checkpointed and moved to the chip overview's task list.

In order to be able to compare the payloads of two or more cores, multiple core views can be opened at once. The user may drag them around to prevent them from lying on top of each other. The popup can also be scaled to fit more of them on the screen, or rotated for when the user wants to look at it from a different angle.

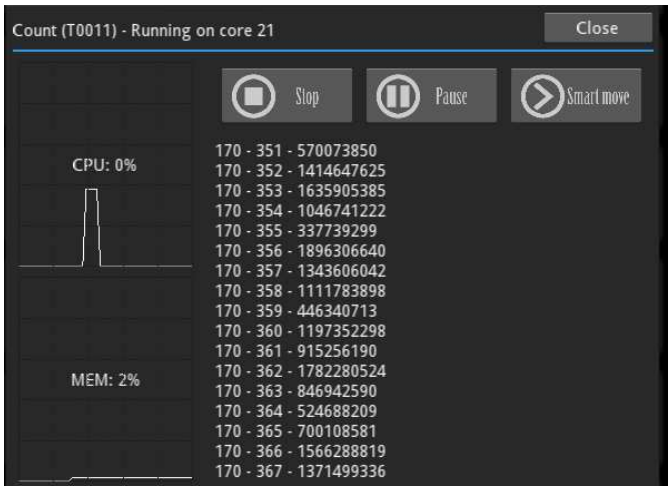


Figure 4: ManyMan's task view, on a task 'Count'.

H. Task view

The detailed task view (see figure 4) looks similar to the core view. Again, the left side of the popup contains information about the CPU and memory usage of the task. On the lower right side of the window, the last 100 lines of the task's output are shown. This number is configurable, but cannot be too large due to Kivy's inefficient way of rendering text. The complete output of a task is written to a file, so that it can be accessed and processed later.

Above the output, the task control buttons are shown. Tapping the stop button will signal the back-end that the task needs to be checkpointed, after which the task will be moved to the chip overview's task list. When a user wants to temporarily pause a task, he can tap the pause button. The back-end will then send a POSIX STOP signal to the task, after which the pause button will be replaced by a resume button. Tapping this button will cause the task to be resumed by sending it the POSIX CONT signal. Finally, tapping the move button will *smart-move* the task to the best available core other than itself.

Figure 5: The task create popup along with the on-screen keyboard.

Just like the core views, multiple task views may be opened at once to compare their performance. It is even possible to have both multiple core and task views open at the same time.

I. Task creation

When tapping the `Add task` button in the main window, a popup will open in which a command and optionally a name can be entered. This popup is shown in figure 5. For multi-touch support, an on-screen keyboard can be used to enter the name and command of the task. A command is either a known shell command or the location of a binary accessible on the cores (for example in the `/shared` directory). After the create button is pressed, the task will be added to the chip overview's task list. A better way of creating a task would be by selecting a binary using a file browser. Unfortunately, the front-end does not run on the SCC's MCPC, which means that one cannot easily open a graphical file browser and navigate to the file.

J. Voltage and Frequency Scaling

Using the `Set Frequency` button, ManyMan is able to set the frequency divider for each tile. Currently, we only change frequencies at voltage domain level, and set the voltage according to the minimal value for that frequency. See table I, obtained from [10], for the corresponding values. As figure 6 shows, one can set the frequency for each of the power domains, or for the chip as a whole. In the main window, the power consumption of the complete chip is visualized in a graph.

Figure 6: Frequency and Voltage settings, easily adjustable.

Frequency divider	Frequency	Voltage
2	800	1.16250
3	533	0.85625
4	400	0.75625
5	320	0.69375
6	267	0.66875
7	229	0.65625
8	200	0.65625
16	100	0.65625

Table I: SCC supported frequencies and required voltage. Based on [10], where frequencies between 100-200 are left out on purpose.

IV. EVALUATION

A. Running latency

In order to be able to stop a task in mid-execution, tasks are started using the `cr_run` command, which might create overhead. In order to test this, experiments have been performed using a program that calculates the sum of some million random numbers. The Unix `time` function has been used to time this program when it is executed both with and without the `cr_run` command. On the SCC, the `time` function is unreliable for measuring the wall-clock time, due to the possibly varying frequencies of the cores where the kernel does not correct for by default. In a period that the frequency does not change, however, this function can be used to measure relative times. We measured an overhead of about 0.2% for this task with a running time of about 55 seconds. In practice, it boils down to less than 0.1 second of overhead per execution.

B. Checkpoint/Restart latency

Checkpointing can create a lot of overhead, as the BLCR library writes the complete state of the process to the filesystem. On the SCC, the only (persistent) filesystem is NFS mounted from the MCPC. This allows easy migration of tasks across cores, but also introduces a large latency for task migration.

Figure 7 shows the latency for checkpointing and restarting processes with increasing memory usage. The testing program

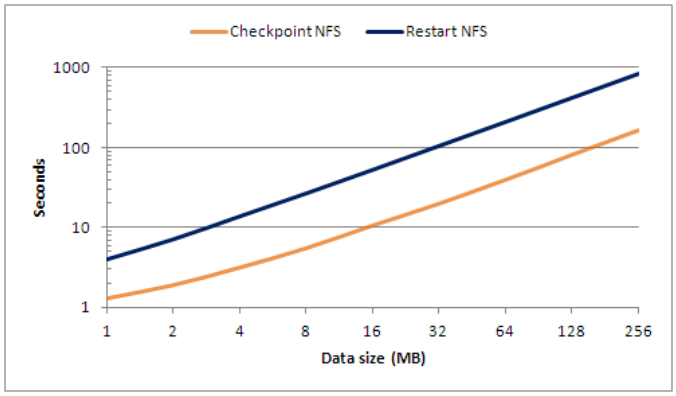


Figure 7: Latency for checkpointing and restarting using the BLCR library on SCC core 0.

that has been used here simply allocates a specified number of megabytes of memory. This memory is then filled with random data, after which the sum of this data is calculated. While calculating the sum, the process will be checkpointed. It is made sure that all requested memory has been allocated and filled with random data at that time. This way, problems with *lazy allocation* of memory pages will not be encountered.

The measurements in this experiment were done by adding a timing mechanism in the BLCR source code. At the beginning and end of the main function, the time stamp counter (TSC) is read and the values are subtracted. To convert to seconds, the resulting value is divided by the core frequency. As expected, the time required for checkpointing scales linearly with the amount of allocated memory.

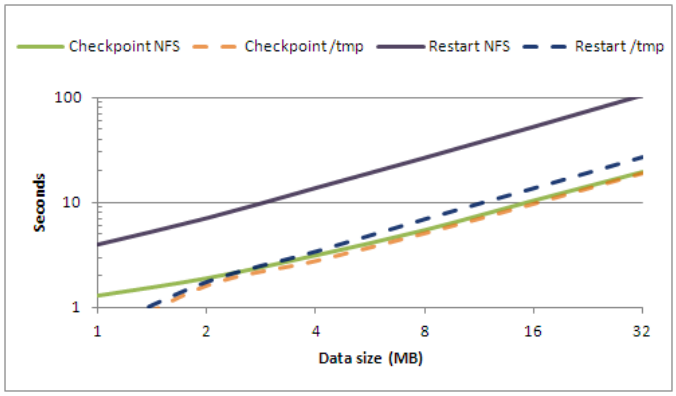


Figure 8: Latency for checkpointing and restarting using the BLCR library on core 0. Writing to NFS compared to writing to RAM.

Besides the test where context files were written to NFS on the MCPC, an additional test has been performed where the context files were written to the filesystem in RAM (`/tmp`). The results of this experiment can be found in figure 8. It shows that the time required for checkpointing can be reduced with approximately 4.5%. When restarting a task from RAM, the difference is much bigger. A task now restarts almost twice as fast, with a speedup of 49.3%. This giant difference is due to the fact that loading data from NFS into RAM is a very time consuming task. When context files are written in RAM,

there is a problem when tasks need to be migrated. As multiple cores do not share their RAM, the context files would have to be copied between cores. This could possibly be done by using *Copy Cores* or shared memory (memory remapping). However, relocating context files would slow down and complicate the restarting process again and probably not be beneficial.

C. Connection Sharing time gain

In order to speed up each access to a core, SSH Connection Sharing is used. We measured the average time it takes to open an SSH connection to SCC core 10, both with and without using Connection Sharing. In order to obtain these results, an `ssh` command was used to open a connection to core 10, on which immediately the `exit` command was executed. We measured an average speedup of 0.61 seconds when using Connection Sharing.

In order to make sure these results were not distorted due to immediately closing the connection using the `exit` command, additional experiments have been performed in which a `sleep` of 10 and 20 seconds has been executed. For these experiments, the average time gains were 0.62 and 0.61 seconds respectively, by which the initial measurement is confirmed.

D. Energy Consumption

We performed a very small power consumption measurement in which we calculate the number of (floating point) operations in an iterative estimation of π. The results of this test can be found in table II. We can go from as low as 21 W at 100 MHz to 110 W at 800 MHz. As we count the number of operations per watt, we see that 320-400 MHz is the most efficient in power consumption. For an idle system, we can easily scale back to 100 MHz and only consume 18 Watts.

Freq.	Volt.	FLOP/s	Power	FLOPs / Watt	Idle
800	1.16250	2232382092	110W	20294383	64W
533	0.85625	1517496998	48W	31614521	30W
400	0.75625	1138837303	35W	32538209	24W
320	0.69375	912457180	28W	32587756	22W
267	0.66875	760936186	25W	30437447	21W
229	0.65625	653551028	24W	27231293	20W
200	0.65625	570407521	23W	24800327	19W
100	0.65625	285427629	21W	13591792	18W

Table II: SCC power consumption

E. Usability test

In order to test the usability of the front-end, a few Computer Science students and a couple of students from non-computer related disciplines were asked to perform a number of tasks. After these tasks had been completed, the students were asked a number of questions about the usability of the software. The tasks that had to be performed and the questions that have been asked can be found in [11], together with a more detailed analysis of the results.

The general opinion of both the Computer Science and the non-Computer Science students was that the tool looked great. They all found the way tasks have to be started very intuitive and really liked the detailed core overview. During these tests, some remarks were made by the participants of which most have been added to the application.

V. CONCLUSION

The proposed application provides any user (either a computer scientist or not) with a total insight of the resource usage on a many-core system like the SCC. The current design of the system is modular, which means that we can easily adapt the font-end to work with a different many-core system than the SCC, but we can also use the back-end for other purposes. The current system may also be usable to manage a cluster of independent (unix) machines with a shared file system, as we currently consider the SCC as a cluster on chip.

The work described in this paper can be extended with some more additional features. Examples of those can be support for MPI tasks including core selection, or automated task scheduling and migration. This scheduler should then be able to automatically adjust voltage and frequency based on the load of the system and possible deadlines. One can also investigate the options for replacing the BLCR library with a more sophisticated cluster management system.

The software created in this project is available for download under the GPL3 license at [12], where we also provide more information about the project, screenshots and a demonstration video.

REFERENCES

[1] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pp. 108–109, February 2010.

[2] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe, "Early experience with the Barrelfish OS and the Single-chip Cloud Computer," in *3rd Many-core Applications Research Community (MARC) Symposium*, KIT Scientific Publishing, September 2011.

[3] Intel Labs, *SCC External Architecture Specification*, revision 1.1 ed., November 2010.

[4] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope, "Efficient memory copy operations on the 48-core intel scc processor," in *3rd Many-core Applications Research Community (MARC) Symposium*, KIT Scientific Publishing, September 2011.

[5] R. Bakker and M. W. van Tol, "Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium* (P. Tröger and A. Polze, eds.), no. 55, Feb. 2012.

[6] "Qnx introduces breakthrough in multi-core visualization tools (http://www.qnx.com/news/pr_2094_1.html)," 2006.

[7] D. Bacon, P. Cheng, D. Frampton, D. Grove, M. Hauswirth, and V. Rajan, "Demonstration: On-line visualization and analysis of real-time systems with tuningfork," in *Compiler Construction* (A. Mycroft and A. Zeller, eds.), vol. 3923 of *Lecture Notes in Computer Science*, pp. 96–100, Springer Berlin / Heidelberg, 2006.

[8] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006.

[9] "Kivy (http://kivy.org)," 2012.

[10] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the intel scc," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 139–149, 2011.

[11] J. van der Woning, "Interactive visualization and dynamic task management of many-core systems. A case-study: The Intel Single-chip Cloud Computer," bachelor's thesis, University of Amsterdam, jun 2012.

[12] J. van der Woning, "ManyMan." Online, http://student.science.uva.nl/~jimivdw/manyman/ [visited jun 2012].